GPUによる処理の高速化

前田利之（まえだ　としゆき）
aipr@e-chan.jp

阪南大学　経営情報学部

(2024.12.13)

ニューラルネットの学習の問題点

- データ量が大きいと、計算時間がかかる
  - ネットワークをデータが順伝搬するのに [ ] の単純な積和計算を行なう
  - 逆伝搬も同様
- CPU パワーには限界があるので高速化には限度がある
- 膨大な数の単純な積和計算→実は、 [ ] の得意分野

ニューラルネットの学習の問題点

- データ量が大きいと、計算時間がかかる
  - ネットワークをデータが順伝搬するのに
    膨大な数の単純な積和計算を行なう
  - 逆伝搬も同様
- CPU パワーには限界があるので高速化には限度がある
- 膨大な数の単純な積和計算→実は、　　の得意分野

ニューラルネットの学習の問題点

- データ量が大きいと、計算時間がかかる
  - ネットワークをデータが順伝搬するのに
    膨大な数の単純な積和計算を行なう
  - 逆伝搬も同様
- CPU パワーには限界があるので高速化には限度がある
- 膨大な数の単純な積和計算→実は、 GPU の得意分野

GPU (Graphics Processing Unit)

- コンピュータゲームに代表されるリアルタイム［　　］に特化したプロセッサ
- 3D CG などに必要な、定形かつ大量の演算を［　　］処理するグラフィックスパイプライン処理性能を重視
- 高速のビデオメモリ（VRAM）と接続され頂点処理およびピクセル処理などの座標変換や陰影計算（シェーディング）に特化したプログラム可能な［　　］

GPU (Graphics Processing Unit)

- コンピュータゲームに代表されるリアルタイム**画像処理**に特化したプロセッサ
- 3D CG などに必要な、定形かつ大量の演算を　　処理するグラフィックスパイプライン処理性能を重視
- 高速のビデオメモリ（VRAM）と接続され頂点処理およびピクセル処理などの座標変換や陰影計算（シェーディング）に特化したプログラム可能な

GPU (Graphics Processing Unit)

- コンピュータゲームに代表されるリアルタイム**画像処理**に特化したプロセッサ
- 3D CG などに必要な、定形かつ大量の演算を**並列パイプライン**処理するグラフィックスパイプライン処理性能を重視
- 高速のビデオメモリ（VRAM）と接続され頂点処理およびピクセル処理などの座標変換や陰影計算（シェーディング）に特化したプログラム可能な

GPU (Graphics Processing Unit)

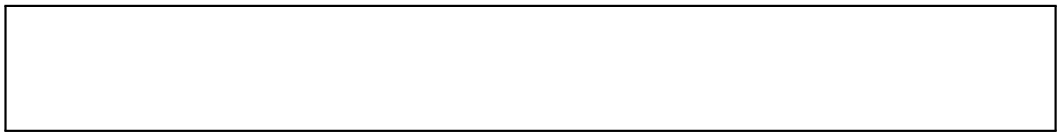
- コンピュータゲームに代表されるリアルタイム**画像処理**に特化したプロセッサ
- 3D CG などに必要な、定形かつ大量の演算を**並列パイプライン**処理するグラフィックスパイプライン処理性能を重視
- 高速のビデオメモリ（VRAM）と接続され頂点処理およびピクセル処理などの座標変換や陰影計算（シェーディング）に特化したプログラム可能な**演算器を多数搭載**

GPU (Graphics Processing Unit)

- CPU よりも並列演算性能にすぐれた GPU のハードウェアを一般的な計算に活用する→
  - 映像出力端子を持たない専用製品もある
  - 深層学習ベースの AI 向けに特化した演算器を搭載したハイエンド製品もある

GPU (Graphics Processing Unit)

- CPU よりも並列演算性能にすぐれた GPU のハードウェアを一般的な計算に活用する→ GPGPU



  - 映像出力端子を持たない専用製品もある
  - 深層学習ベースの AI 向けに特化した演算器を搭載したハイエンド製品もある

GPU (Graphics Processing Unit)

- CPU よりも並列演算性能にすぐれた GPU のハードウェアを一般的な計算に活用する→ GPGPU (General Purpose GPU)
  - 映像出力端子を持たない専用製品もある
  - 深層学習ベースの AI 向けに特化した演算器を搭載したハイエンド製品もある

GPU (Graphics Processing Unit)

- 多数のコアを独立に使える処理が得意
  - ニューラルネットの伝搬処理はまさにこれ
  - 独立じゃない→順序依存な処理とは…例えば
    A=1; B = A+1; C = A+B;
    のような場合、この順番で処理を進めないと結果が出ない、のようなこと

GPU の代表的製品

- NVIDIA ... 圧倒的シェア
  - GeForce シリーズ (GTX/RTX,... のちに Quadro)
  - TESLA シリーズ (GPGPU のハイエンド)
  - Jetson シリーズ (Nano, Xavier,... 単体で動くコンピュータシステム。学習用 and/or エッジコンピューティング)
- AMD, Intel,... 後塵を拝している (?!)

CUDA

- NVIDIA 製 GPGPU を使うためのライブラリ
- [　　] で書かれている（ハードウェアに近い処理が必要）
- うまく使えば、[　　]計算で圧倒的な処理能力を発揮できる

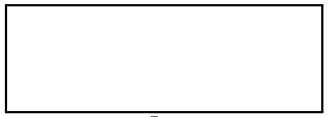
CUDA

- NVIDIA 製 GPGPU を使うためのライブラリ
- C/C++ で書かれている（ハードウェアに近い処理が必要）
- うまく使えば、 計算で圧倒的な処理能力を発揮できる

CUDA

- NVIDIA 製 GPGPU を使うためのライブラリ
- C/C++ で書かれている（ハードウェアに近い処理が必要）
- うまく使えば、行列の積和計算で圧倒的な処理能力を発揮できる

CuPy

- python から cuda を使えるようにする
  
-  とある程度の互換性があるので、運が良ければ  を cupy と書きかえるだけで GPGPU の恩恵にあずかることが出来る、が、微妙な違いに苦労することも多い…（後述）

CuPy

- python から cuda を使えるようにする
  ライブラリ
- [ ] とある程度の互換性があるので、運が良ければ [ ] を cupy と書きかえるだけで GPGPU の恩恵にあずかることが出来る、が、微妙な違いに苦労することも多い…（後述）

CuPy

- python から cuda を使えるようにする
  ライブラリ
- numpy とある程度の互換性があるので、運が良ければ [ ] を cupy と書きかえるだけで GPGPU の恩恵にあずかることが出来る、が、微妙な違いに苦労することも多い…（後述）

CuPy

- python から cuda を使えるようにする
  ライブラリ
- numpy とある程度の互換性があるので、運が良ければ numpy を cupy と書きかえるだけで GPGPU の恩恵にあずかることが出来る、が、微妙な違いに苦労することも多い…（後述）

CuPy

- インストール時には cuda 関連のインストールを伴なうので、結構な時間がかかる
- ⇐ C/C++ で書かれている部分があり、それらをコンパイルする必要があるから
- ※注意※ Colab. では毎回リセットされるので毎回インストールが必要だが、自宅 PC にインストールする場合は最初の１回だけでよい
- ※さらに注意※今 (2023.12) では Colab. にデフォルトでインストールされているので作業不要になった

cupy を使った畳み込みニューラルネット

1. まず、 myc.zip のリンク先を解凍し（nantoka_c.py がいくつかできる）そのファイルを Colab. の mymodules の中に保存する
2. それぞれの nantoka_c.py は元のファイルが numpy をインポートしているところを

   import cupy as np

   としている。さらに、simple_convnet_c.py では

   acc += nup.sum(np.asnumpy(yy) == tt)

   と、また trainer_c.py では

   x_batch = self.x_train[batch_mask.get()]

   t_batch = self.t_train[batch_mask.get()]

   と書きかえられている

cupy を使った畳み込みニューラルネット（続き）

3. 「ライタイム」メニューの「ランタイムのタイプを変更」で GPU を選ぶ (重要!)
4. その上で、con_nn_c.py （以下はコード）のリンク先を Colab. にコピー＆ペーストして実行する

```
# coding: utf-8
from google.colab import drive
drive.mount('/content/drive',force_remount=True)
bd = 'drive/My Drive/Colab Notebooks/mymodules'
import sys, os
sys.path.append(bd)  # mymodules 対応
import numpy as np # ここは cupy にしないで良い（！）
import matplotlib.pyplot as plt
# !pip install cupy # cupy のインストールは不要になった
```

（…以降に続く前に処理が必要）

畳み込みニューラルネットを使った学習

（前頁からの続き）

```
# cupy 対応の _c のモジュールに変更（!!）
from mnist_c import load_mnist
from simple_convnet_c import SimpleConvNet
from trainer_c import Trainer
# データの読み込み
(x_train, t_train), (x_test, t_test) = load_mnist(flatten=False)
max_epochs = 20
network = SimpleConvNet(input_dim=(1,28,28), \
     conv_param = {'filter_num': 30, 'filter_size': 5, 'pad': 0, \
    'stride': 1}, hidden_size=100, output_size=10, weight_init_std=0.01)
trainer = Trainer(network, x_train, t_train, x_test, t_test, \
     epochs=max_epochs, mini_batch_size=100, \
     optimizer='Adam', optimizer_param={'lr': 0.001},\
     evaluate_sample_num_per_epoch=1000)
trainer.train()
```

（…以下、con_nn と同じなので略）

cupy を使った畳み込みニューラルネット

5. !pip install cupy
   は、Colab. 内で pip コマンド = Python のライブラリをネット越しにインストールできるが、今は不要（前述）なので↓
6. (30 分以上待たされインストールは出来る、ではなく) 待たされないが、 cupy のライブラリの実行時に失敗する

```
(..snip)
     79         for size, (left, right) in zip(array.shape, pad_wid
     80     )
---> 81     padded[original_area_slice] = array
     82
     83     return padded, original_area_slice
```

cupy を使った畳み込みニューラルネット

7. pad.py 内 で（非互換なところで）怒られているので、リンクをクリックで開いて

```
padded[original_area_slice] = array
```

を、

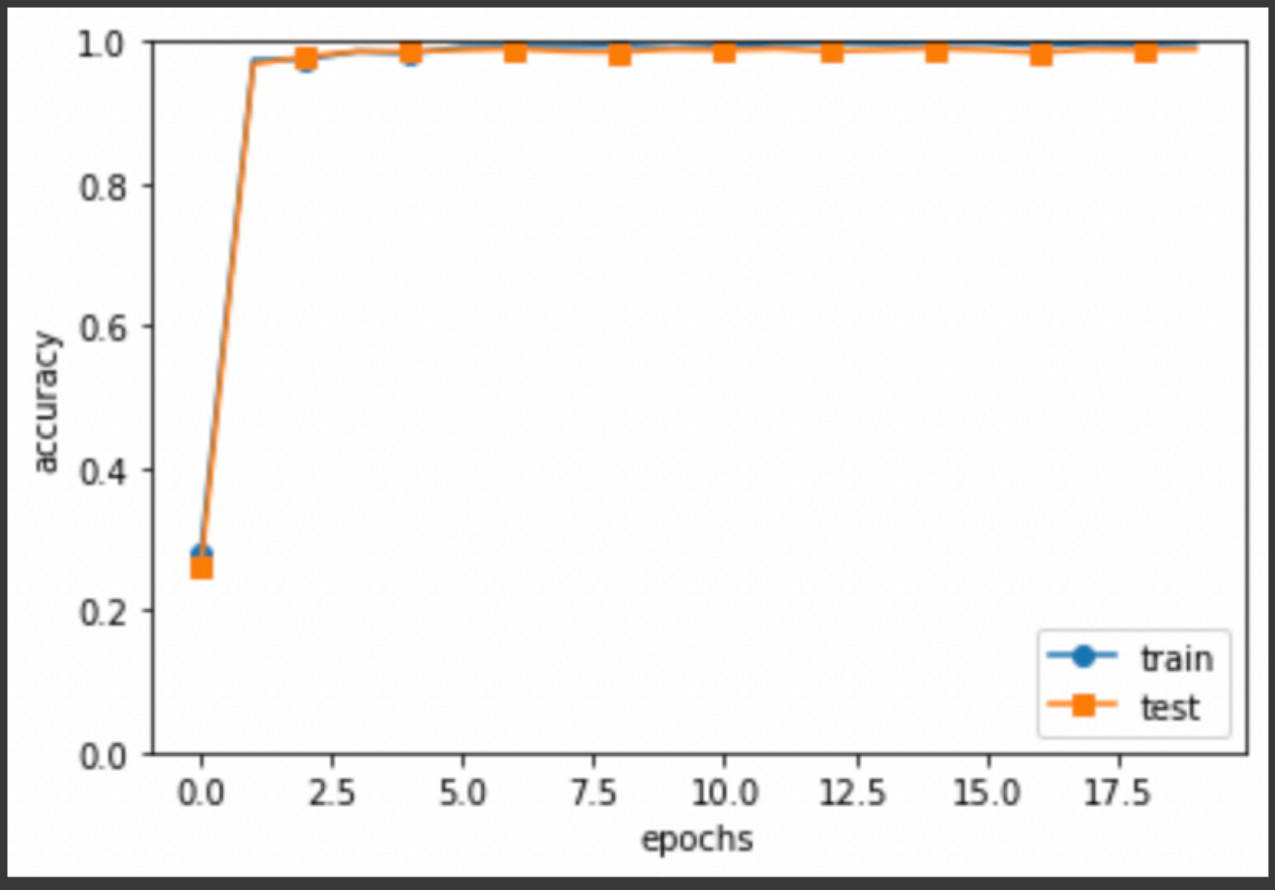
```
padded[original_area_slice] = \
  cupy.asarray(array) # 本当は１行
```

に書き換える

8. 「ライタイム」メニューの「ランタイムの再起動」を実行する (重要!)
9. コードエリアの左にある▲（ほんとは右向き）をクリックして実行しなおす

実行結果

```
=== epoch:19, train acc:0.996, test acc:0.988 ===
=== epoch:20, train acc:0.998, test acc:0.989 ===
============== Final Test Accuracy ==============
test acc:0.9872
Saved Network Parameters!
```



実行結果

- CuPy のインストール時間を除けば、20 エポックをだいたい 6 分前後で終わらせる（はず）
- 普通の numpy では、10 エポックで半時間ほどだった→ 20 エポックだと 60 分ほど（？）
- → cupy に切り替えるだけで、約 [ ] の高速化が可能となった (!!)

実行結果

- CuPy のインストール時間を除けば、20 エポックをだいたい 6 分前後で終わらせる（はず）
- 普通の numpy では、10 エポックで半時間ほどだった→20 エポックだと 60 分ほど（？）
- → cupy に切り替えるだけで、約 10 倍の高速化が可能となった (!!)

おまけ: cython

- そもそも python 自体が結構複雑な処理をしている
- → 
- なので、python を  =軽い言語に変換して、コンパイル処理しておくと劇的に速くなる場合がある ⇒ cython
- numpy でも活用できる

おまけ: cython

- そもそも python 自体が結構複雑な処理をしている
- → 遅い（重い）
- なので、python を [　　] =軽い言語に変換して、コンパイル処理しておくと劇的に速くなる場合がある ⇒ cython
- numpy でも活用できる

おまけ: cython

- そもそも python 自体が結構複雑な処理をしている
- → 遅い（重い）
- なので、python を C/C++ =軽い言語に変換して、コンパイル処理しておくと劇的に速くなる場合がある ⇒ cython
- numpy でも活用できる

おまけ: cython (書きかた)

- 関数は python のものをそのまま持ってきて、少し中身を変更する
- int 型に固定したい変数があれば、 [ ] x で固定する
- 関数を [ ] で定義すると python からは呼び出せないが cython の他の関数からだけ見えるので、その分早くなる
- [ ] で関数を定義すると、cython から見る用のものと、python から呼び出す用のものが両方作成される（はず）

おまけ: cython (書きかた)

- 関数は python のものをそのまま持ってきて、少し中身を変更する
- int 型に固定したい変数があれば、 `cdef int` x で固定する
- 関数を [　　] で定義すると python からは呼び出せないが cython の他の関数からだけ見えるので、その分早くなる
- [　　] で関数を定義すると、cython から見る用のものと、python から呼び出す用のものが両方作成される（はず）

おまけ: cython (書きかた)

- 関数は python のものをそのまま持ってきて、少し中身を変更する
- int 型に固定したい変数があれば、 cdef int x で固定する
- 関数を cdef で定義すると python からは呼び出せないが cython の他の関数からだけ見えるので、その分早くなる
- [ ] で関数を定義すると、cython から見る用のものと、python から呼び出す用のものが両方作成される（はず）

おまけ: cython (書きかた)

- 関数は python のものをそのまま持ってきて、少し中身を変更する
- int 型に固定したい変数があれば、 `cdef int` x で固定する
- 関数を `cdef` で定義すると python からは呼び出せないが cython の他の関数からだけ見えるので、その分早くなる
- `cpdef` で関数を定義すると、cython から見る用のものと、python から呼び出す用のものが両方作成される（はず）

おまけ: cython (実践)

1. cython_m.ipynb （以下はコード）のリンク先を Colab. にコピーして実行する

```
!pip install Cython #==3.0.0a11
%load_ext Cython # これで cython が使えるようになる
```

（…以降に続く前に処理が必要）

おまけ: cython (実践)

（前頁からの続き）

```
%%cython # cython を使う宣言 @ Colab.
def csumtest(int n):
  cdef int i
  cdef long sum
  print("Cython sumtest.sumtest({}) が呼ばれた"\
        .format(n)) # 本当は１行
  sum=0
  for i in range(1,n+1):
    sum+=1
  return sum
```

おまけ: cython (実践)

（前頁からの続き）

```
def sumtest(n):
  sum=0
  print("Python sumtest.sumtest({}) が呼ばれた"\
        .format(n)) # 本当は１行
  for i in range(1,n+1):
    sum+=1
  return sum

import time

n=1000*1000*200
```

おまけ: cython (実践)

（前頁からの続き）

```
btime=time.time()
s=sumtest(n)
etime=time.time()
gtime=etime-btime
print("sumtest({})={}経過時間={}秒".format(n,s,gtime))
btime=time.time()
s=csumtest(n)
etime=time.time()
gtime=etime-btime
print("csumtest({})={}経過時間={}秒".format(n,s,gtime))
```

おまけ: cython (実践)

（実行結果）

```
Python sumtest.sumtest(200000000) が呼ばれた
sumtest(200000000)=200000000 経過時間=13.605504035949707 秒
Cython sumtest.sumtest(200000000) が呼ばれた
csumtest(200000000)=200000000 経過時間=0.00012516975402832 03 秒
```

おまけ: cython (実践の考察)

- まさに、桁違いの高速化が可能となっている
- ただし、C コンパイラの最適化、仮想マシンの構成（キャッシュの大きさ）などにも依存する（はず…詳細略）
- あと、事前の準備のコストはかかっていることに注意

おしまい

質問・コメント等あれば、何でもお気軽に
aipr@e-chan.jp に
メールください！

【注意】申し訳ないですが *HInT* の「連絡」は溜めてしまうことが多いのでリプライが遅れがちです。出来るだけ普通のメールで送ってください。